Bubble Sort and its Variants with OpenMP

Sidhartha Ananthula - 21MCME08

November, 2024

Abstract

This project implements a parallel bubble sort algorithm and an improved parallel bubble sort algorithm using OpenMP. The performance of these parallel algorithms is compared and analyzed.

1 System Configuration

The experiments were performed on the following system:

- **Laptop:** MSI GF63
- **Operating System:** openSUSE Leap 15.6
- **CPU:** Intel i5-9300H (8 cores) @ 4.100GHz

2 Code Overview

The code is written in C and uses OpenMP, an API for shared-memory MIMD programming. It has three source files: `main.c`, `sort.c`, and `inout.c`. Each file and its functionalities are described below:

2.1 `main.c`

This file contains the program's entry point (`main()` function):

- Accepts the number of threads and size of data as command-line arguments.
- Prompts the user to select a sorting algorithm to execute.
- Executes the selected algorithm on randomly generated integer input and displays the execution time.
- Saves sorted data to `output.csv` and execution results to `results.csv`.
- Generated input data is saved in `input.csv`.

2.2 `sort.c`

This file contains the implementations of sorting algorithms. Each function takes a pointer to the data and its size as input.

2.2.1 `bubble_sort()`

- Implements the basic Bubble Sort algorithm.

2.2.2 `odd_even_transposition_sort()`

- Implements the Odd-Even Transposition Sort algorithm, as described in [Pac22].

2.2.3 `omp_odd_even_transposition_sort_01()` and `omp_odd_even_transposition_sort_02()`

- Implements the parallel Odd-Even Transposition Sort algorithms using OpenMP, based on the approach in [Pac22].

2.2.4 `shell_sort()`

- Implements the Shell Sort algorithm [She59].

2.2.5 `omp_shell_sort()`

- Implements a parallel version of Shell Sort [Qui88].
- Eliminated loop-carried dependencies by modifying the previous `shell_sort()` algorithm.
- Uses a gradual halving gap sequence [ASA07].
- Each thread sorts sublist(s) of elements, with a reducing gap sequence, using a modified insertion sort.

2.2.6 `omp_shell_sort_02()`

- Implements a parallel version of Shell Sort using the Hibbard Sequence [Qui88].
- The Hibbard Sequence is defined as:

$$g_i = 2^i - 1, \quad i = 1, 2, \ldots, \lfloor \log_2(n) \rfloor$$

2.3 `inout.c`

2.3.1 `generate_input()`

- Generates random integers in the range $[1, 0.75 \times \text{size}]$.
- Saves the data to a CSV file.

2.3.2 `read_input()`

- Reads integer data from a CSV file into a pointer variable.

2.3.3 `write_output()`

- Writes sorted data to a CSV file.

2.3.4 `write_result()`

- Records the algorithm, data size, thread count, and execution time in `results.csv`.
- This data is used for performance analysis.

3 Results and Performance Analysis

The initial idea to use a Parallel Shell Sort algorithm to improve performance was inspired from [GGKK03].

The results generated in `results.csv` allow for detailed performance analysis. Multiple runs with varying data sizes and thread counts provide insights into scalability and efficiency.

Each algorithm is run:

- 100 times for the input sizes 1000, 5000, & 10000.
- 50 times for the input sizes 20000, 40000, 60000, & 80000.
- 5 times for the input sizes 100000, & 200000.
- 1 time for the input sizes 300000, 400000, & 500000.
- 5 times for the previous input sizes in the case of parallel shell sort.

Each of the above is run with 1, 2, 4, 8, & 16 threads, in the case of a parallel algorithm.

3.1 Serial Algorithms Performance

Input Size	Execution Time (seconds)		
	Bubble Sort	**Odd-Even Transposition Sort**	**Shell Sort**
1000	0.0024028083	0.0022502793	0.0017570042
5000	0.0616279375	0.0558655443	0.0490854781
10000	0.2706816557	0.2494372873	0.1955941217
20000	1.1772869826	0.9788865098	0.7705120767
40000	4.8420622367	3.9837998999	3.0417494783
60000	11.0801380679	9.1022774343	5.6704730329
80000	19.7361797654	16.1064127587	12.1952081564
100000	30.8882639482	25.1810321274	16.322841126
200000	124.6022764286	100.231252651	65.7987660648
300000	280.724216636	226.053211357	174.246453292
400000	495.188320413	400.426751025	263.5853297
500000	776.822895897	634.396241299	394.785282869

Table 1: Execution Times for Serial Algorithms

From Table 1 the following can be inferred:

- Shell Sort consistently outperformed Bubble Sort and Odd-Even Transposition Sort for all input sizes, and this is because of its optimized gap-based comparison strategy.
- However, with increase in input size, all 3 serial algorithms show high growth in execution time and hence displays the weakness of serial processing for large data.

3.2 Parallel Algorithms Performance

Speedup Formula

The speedup $S(p)$ for an algorithm using p threads/processes is given by:

$$S(p) = \frac{T(1)}{T(p)}$$

where:

- $T(1)$ is the execution time using 1 thread/process.
- $T(p)$ is the execution time using p threads/processes.

- The above Speedup Formula compares the performance improvement when increasing the number of threads from 1 to p.
- This is used to calculate the Max Speedup in Tables 2, 3, and 4, to indicate the maximum speedup achieved by that algorithm for a given input size with the use of 2, 4, 8, or 16 threads.

3.2.1 Parallel Odd-Even Transposition Sort

Input Size	**Execution Time (seconds)**					**Max Speedup**
	1 Thread	**2 Threads**	**4 Threads**	**8 Threads**	**16 Threads**	
1000	0.0043995729	0.0056291712	0.0055408402	0.0127549767	0.0483956481	0.79
5000	0.0717712518	0.0281106742	0.0247649248	0.0297316774	0.1363136705	2.90
10000	0.224461884	0.1094497986	0.0678424353	0.2360771161	0.0863311228	3.31
20000	0.9341625587	0.4431300019	0.2716702308	0.2743011727	0.6438009419	3.44
40000	3.9170673179	1.8943826517	1.0648950346	0.9877953404	1.7555050802	3.97
60000	8.9916148756	4.3717881219	2.5857709952	2.3413656621	4.4660238177	3.84
80000	16.5034890454	8.482658389	5.7258598321	5.4984473008	6.2940302104	3.00
100000	25.6711403608	12.8538411842	7.5483351732	7.6591181798	9.2750615252	3.40
200000	98.8217707366	52.5538378264	34.4693835264	30.6972018662	32.5410806188	3.22
300000	221.758048915	118.154205375	75.435695432	66.757274992	71.407025189	3.32
400000	404.49093609	210.192399704	134.444006517	115.949780129	123.78182887	3.49
500000	643.211717774	327.143383715	209.064745803	185.268861971	190.543492433	3.47

Table 2: Execution Times for Parallel Odd-Even Transposition Sort

From Table 2 the following can be inferred:

- Parallel Odd-Even Transposition sort exhibits a good improvement from the serial version of the same, as the number of threads increases.
- But this still suffers from the high increase in execution time as the input size is increased.
- Another observation that can be made is, for a given input size, the execution time increased from 8 to 16 threads. This is because of the overhead incurred from context-switches among the threads upon trying to use more threads than available cores.
- An improvement with more threads might be visible if executed on a more powerful machine with many more cores available.

3.2.2 Parallel Shell Sort - Gradual Halving Gap Sequence

Input Size	**Execution Time (seconds)**					**Max Speedup**
	1 Thread	**2 Threads**	**4 Threads**	**8 Threads**	**16 Threads**	
1000	0.0001160222	0.0001258516	0.0001378102	0.0068138312	0.0005858502	0.92
5000	0.0007776907	0.0005769586	0.0005610709	0.0098667189	0.000895235	1.39
10000	0.0015238828	0.0009904061	0.000743426	0.009255718	0.001276754	2.05
20000	0.003438078	0.0020642728	0.0014414798	0.0071212781	0.0025664078	2.39
40000	0.007596714	0.0043324886	0.0030763631	0.010429079	0.0036892813	2.47
60000	0.012006041	0.0075229596	0.0054520347	0.0125844578	0.0052426324	2.29
80000	0.0172282489	0.0099466387	0.0071571829	0.0198035278	0.0069511669	2.48
100000	0.024062812	0.0128602074	0.009147252	0.0215467524	0.0091316722	2.64
200000	0.0546604566	0.0291685302	0.0194057934	0.028510189	0.0191293724	2.86
300000	0.0823413072	0.0437161172	0.0328254946	0.113351331	0.024730842	3.33
400000	0.1154088756	0.060606625	0.0438252642	0.107718486	0.0440991242	2.63
500000	0.1362702092	0.0795114286	0.0540033234	0.2975018922	0.046197056	2.95

Table 3: Execution Times for Parallel Shell Sort (Gradual Halving gap sequence)

From Table 3 the following can be inferred:

- Parallel Shell Sort exhibits a significant improvement in execution time, even for a huge input size of 500,000.
- This algorithm is highly amenable to parallelization and much more efficient to sort huge data than Parallel Odd-Even Transposition sort.
- The overhead when using 16 threads is not observed in Parallel Shell Sort because they are being executed so fast and not much overhead is incurred from context switches.

3.2.3 Parallel Shell Sort - Hibbard Gap Sequence

Input Size	Execution Time (seconds)					Max Speedup
	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	
1000	0.0002227027	0.0002270883	0.0001796288	0.0076201447	0.0006867567	1.24
5000	0.0008005272	0.0005889769	0.0004748321	0.007167066	0.0010458463	1.69
10000	0.0018094457	0.0011815263	0.000940152	0.0079271553	0.0014104779	1.92
20000	0.0041917461	0.0025645993	0.0018749088	0.0138917585	0.0023091777	2.24
40000	0.0093606897	0.0055948565	0.003891705	0.011296769	0.0036881603	2.54
60000	0.0147576036	0.0081541226	0.0054454264	0.0094921519	0.0053113553	2.78
80000	0.0197075926	0.0118485022	0.0080760815	0.009757968	0.007586132	2.60
100000	0.0284322196	0.014945354	0.0099236306	0.0078345594	0.0089508908	3.63
200000	0.0586942048	0.0326963984	0.020659978	0.0150368526	0.0172142492	3.90
300000	0.0954365884	0.0501971524	0.0363003164	0.1515741846	0.029540666	3.23
400000	0.1323158122	0.0746978842	0.048753065	0.1581163424	0.040731122	3.25
500000	0.1731763882	0.0999991686	0.0663707694	0.1987161396	0.050199827	3.45

Table 4: Execution Times for Parallel Shell Sort (Hibbard gap sequence)

From Table 4 the following can be inferred:

- This version of Parallel Shell sort provided similar results to the previous version with a significant improvement in execution times.
- The previous version performs, not much but, slightly better than this in many cases, as it was suggested an Enhanced Shell Sort [ASA07].

3.3 Graphical Analysis

3.3.1 Plot between Parallel Shell Sort and Parallel Odd-Even Transposition Sort

Here we compare the performance of Parallel Shell Sort and Parallel Odd-Even Transposition Sort. The plot shows the time complexity of both algorithms under various input sizes using 8 threads. It turns out that Parallel Shell Sort is highly efficient.

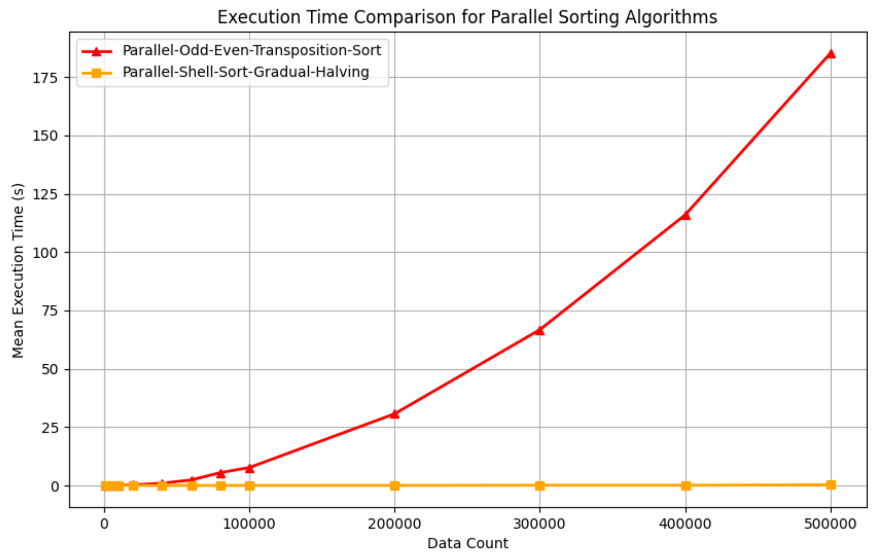

Figure 1: Comparison of Parallel Shell Sort and Parallel Odd-Even Transposition Sort

Speedup plot for the above:

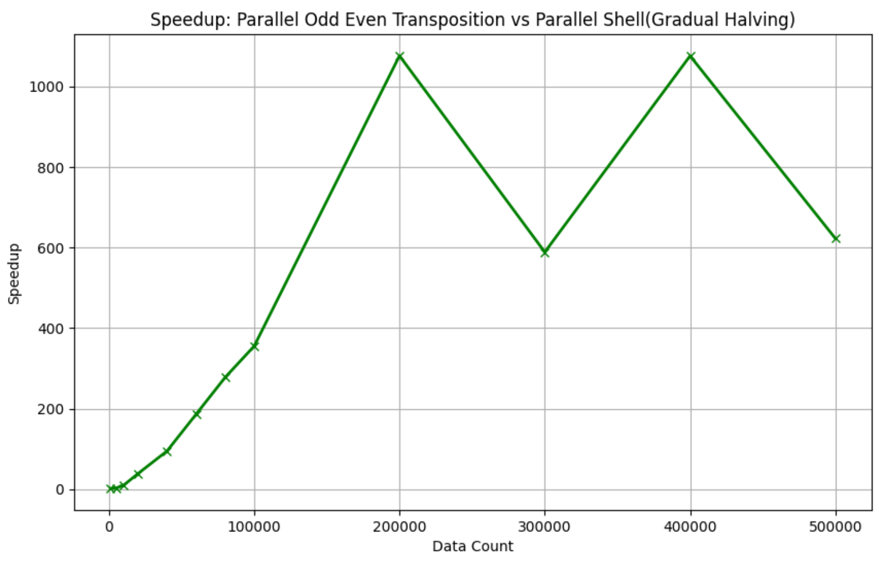

Figure 2: Speedup of Parallel Odd-Even Transposition Sort vs Parallel Shell Sort

3.3.2 Plot between Odd-Even Transposition Sort and Parallel Odd-Even Transposition Sort

This plot shows the comparison between the classic Odd-Even Transposition Sort and its parallel variant run with 8 threads. But neither are scalable.

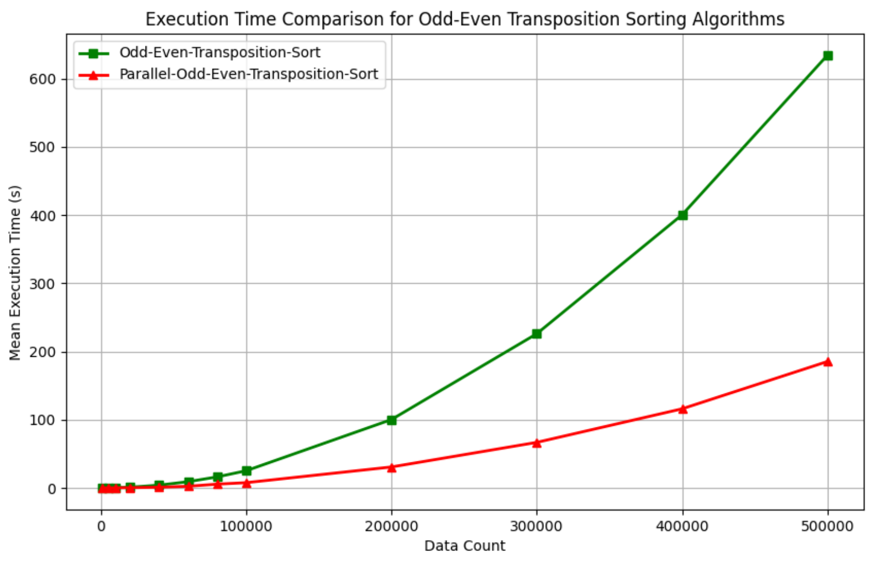

Figure 3: Comparison of Odd-Even Transposition Sort and Parallel Odd-Even Transposition Sort

Speedup plot for the above:

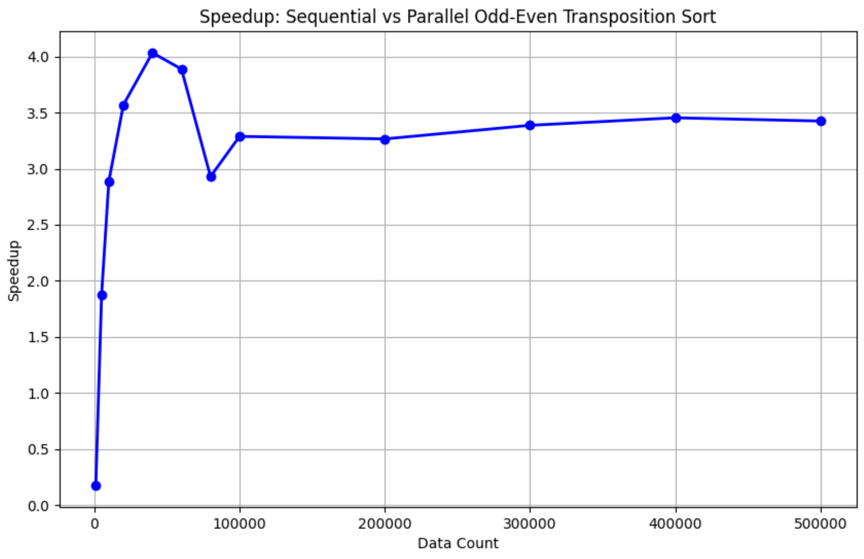

Figure 4: Speedup of Odd-Even Transposition Sort vs Parallel Odd-Even Transposition Sort

3.3.3 Plot between Shell Sort and Parallel Shell Sort variants

This figure compares the performance of the sequential Shell Sort algorithm and its parallel implementations with 8 threads. It highlights how parallelization enhances the algorithm's high efficiency and scalability with larger datasets.

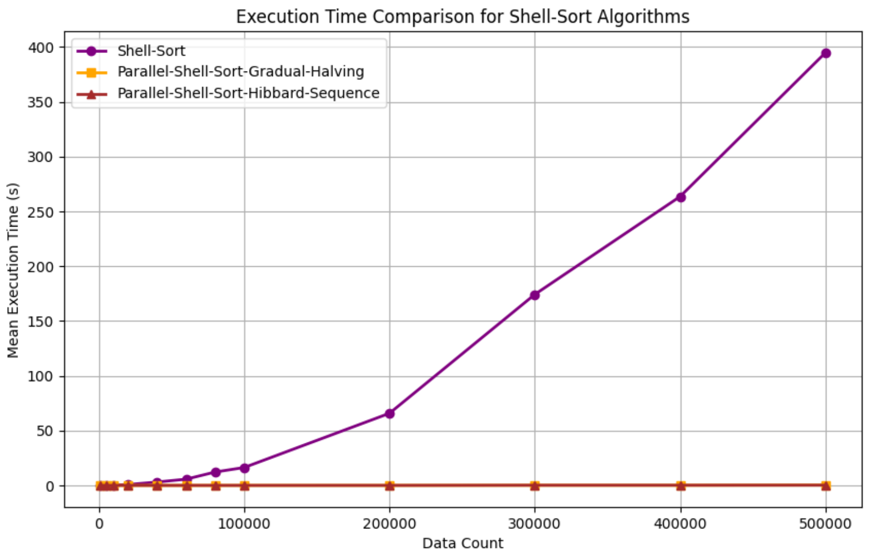

Figure 5: Comparison of Shell Sort and Parallel Shell Sort

Speedup plot for the above:

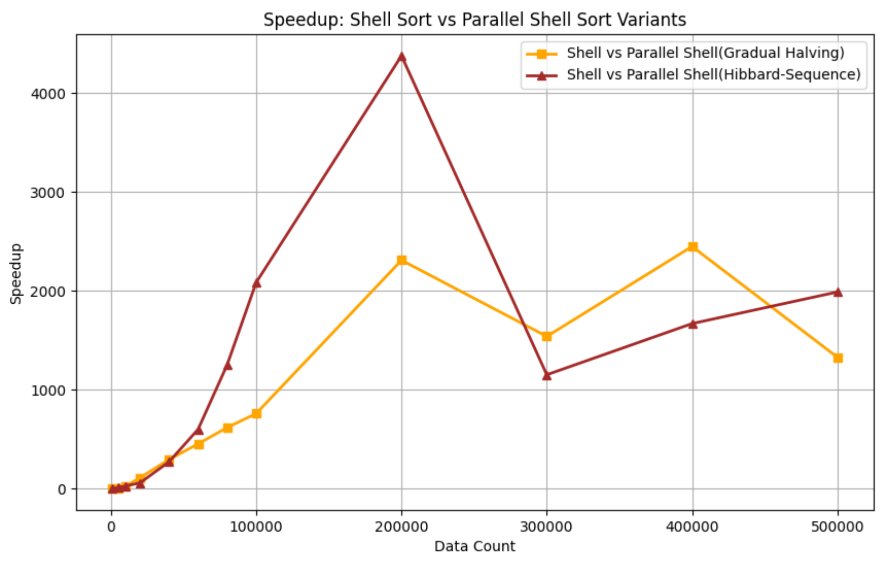

Figure 6: Speedup of Shell Sort vs Parallel Shell Sort variants

References

[ASA07] Tanveer Afzal, Basit Shahzad, and Salman Aslam. Enhanced shell sort algorithm. In *International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol:1, No:3, 2007*, 02 2007.

[GGKK03] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.

[Pac22] Peter S. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2022.

[Qui88] Michael J. Quinn. Parallel sorting algorithms for tightly coupled multiprocessors. *Parallel Computing*, 6(3):349–357, 1988.

[She59] D. L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, July 1959.